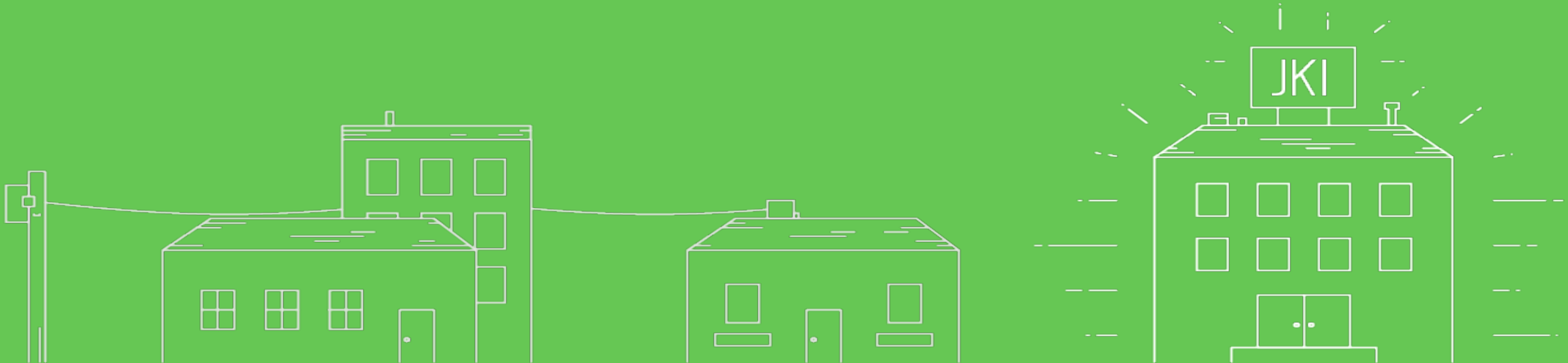
JKI

Future Proofing Your Software

How to Leave a Legacy Without Leaving Legacy Software

CLA Summit Americas
09/25/2019



Agenda

- What is “Legacy Software”?
- I: Design Phase, decisions that improve maintainability
- II: Maintenance Phase, what causes software to grow more complex and how to avoid it
    - Specific examples of how “good” code goes “bad”
- III: Successful Teams, non-technical related aspects of legacy
- Conclusions

Takeaway Goals

- Open the discussion on what makes code “legacy code” and how it gets created.
- Reflect on changes we can make everyday to improve the quality of our codebases.

Key sources of inspiration for this presentation

- “Working Effectively with Legacy Code” - Michael Feathers
- Tech Done Right Podcast: “Avoiding Legacy Code with Michael Feathers”
- Martin Fowler on Refactoring and Technical Debt

What is Legacy Software?

There are many working definitions of what legacy software is. They are never associated with positive qualities.

Just ask some software developers what "Legacy" means to them..

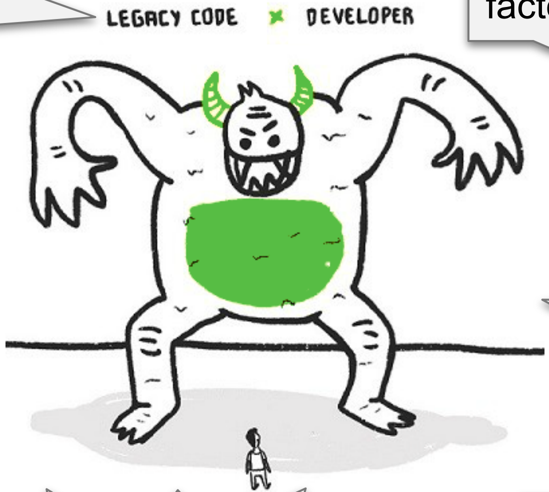
It's code that inspires "terror, disappointment, and disgust"

It's Old

It's poorly written, not modular and not well factored.

It is **any** code base not covered by unit tests

It cannot be extended without massive effort



It's code you'd rather replace than work with

The author has left the company and nobody understands how it works.

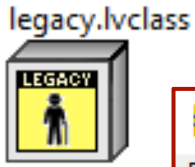
It uses deprecated technologies

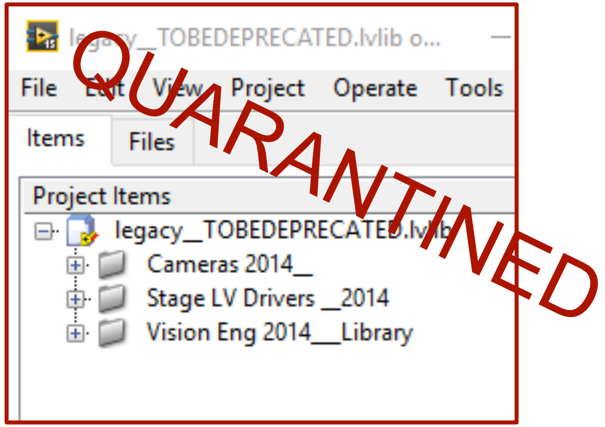
Gradual modifications caused well-written code to evolve into a complex monster

Common Denominator

Legacy software **is software that other developers are afraid to change**

- Changing the software presents a **high risk** due to a variety of factors:
  - Not well designed or maintained
  - Not unit tested
  - The code is too complex
  - Not properly documented
  - Technologies or tools are deprecated
  - **… <Insert your own complaints here>**

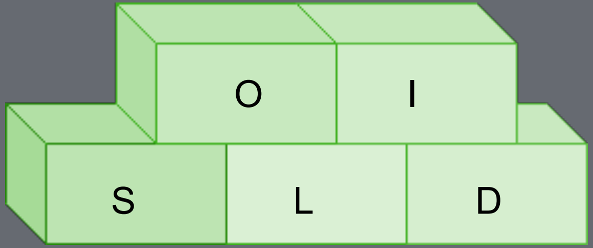




How can we write software so that other developers are **not afraid** to make changes later?

It is not just about the initial design

I: Design Phase



What proactive design decisions can we make ***at the beginning*** *of a project to improve ease of maintainability in the future?*

Sometimes “Easy” is better than “Right”

Keep the Design Simple

- YAGNI!

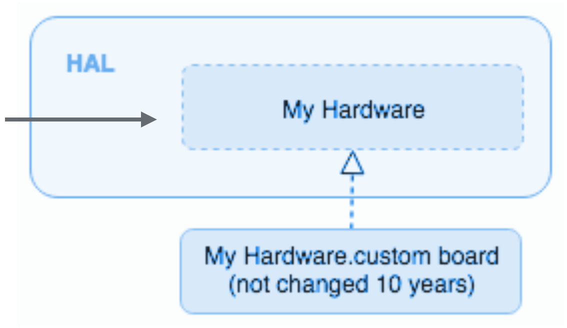
  Sometimes flexibility/scalability is sacrificed for maintainability

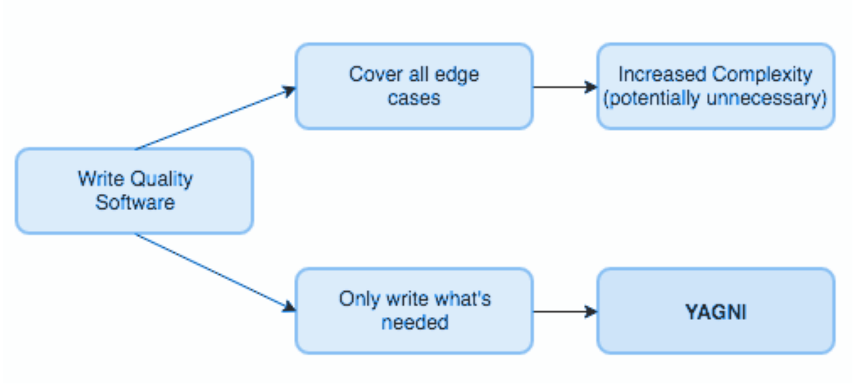
- Design for ease of maintainability first

  Path length to perform an operation is one measure of complexity

  Long path = Complex code (too many layers increases complexity)

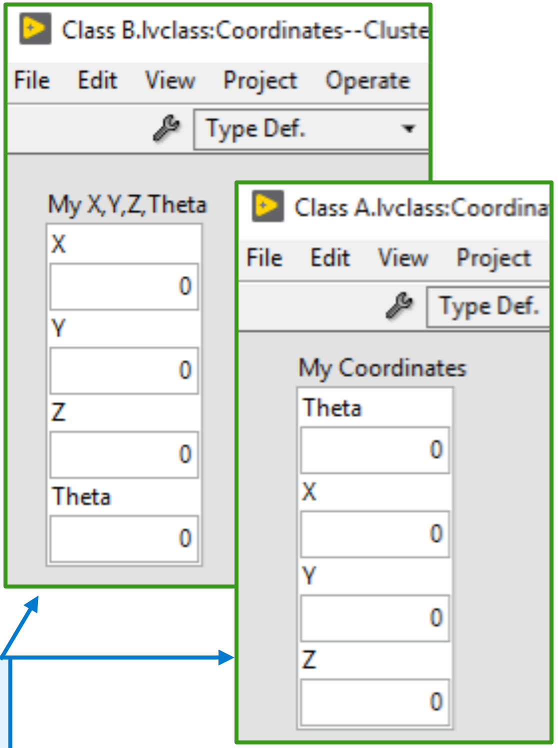
You **A**ren’t
Gonna **N**eed **I**t!





Manage Shared Resources Appropriately

- Proper management of shared resources will prevent unintended coupling
  - Typedefs
    - Only belong as class members if exclusive to class
    - Scope internal typedefs as protected or private
    - Put all other typedefs in a common/shared location.
  - Interlocks, Path Management, Calibration, Etc.
    - Prevent dependencies between otherwise unrelated subsystems
    - May require a dedicated arbitrator



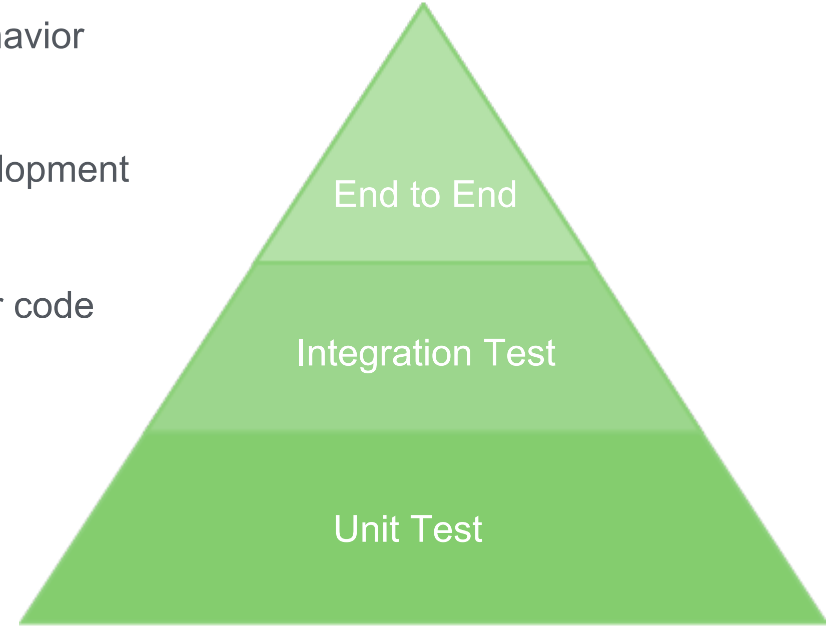
Two classes which own almost the same data structure.. only ordering in the cluster is different. This can cause a big issue at the class boundary.

Michael Feathers: “legacy code is simply code without tests”

Add Unit Testing Early On

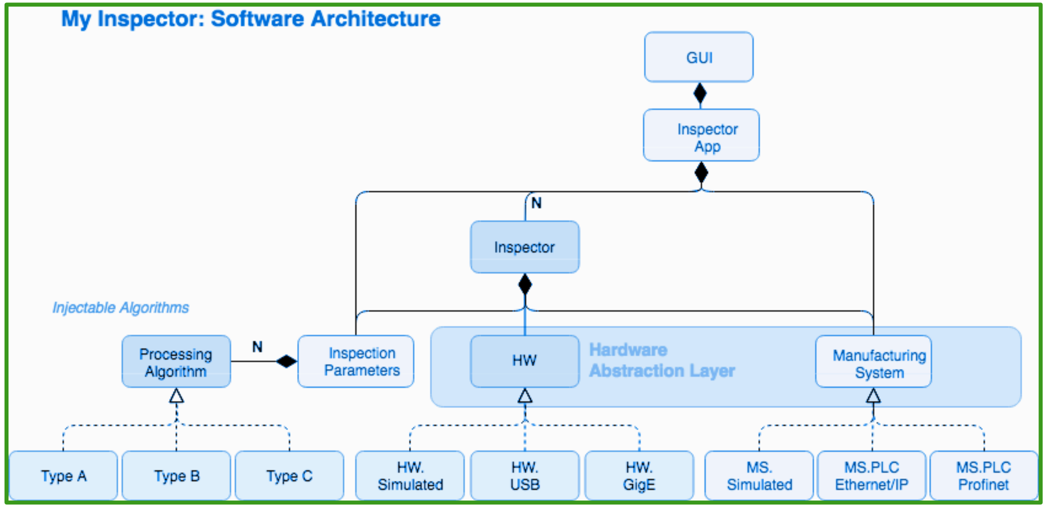
- Tests are a safety net for catching unintentional changes in code behavior
  - **Unit tests reduce fear for developers to make changes**
  - Easy to understand if changes impacted other behavior

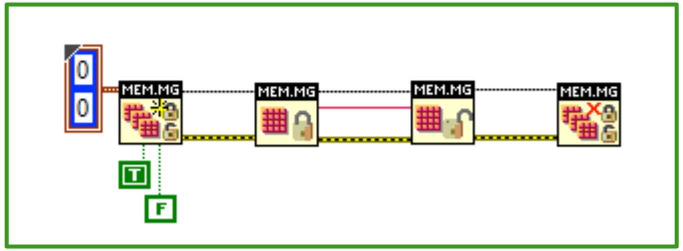
- It is painful to add test coverage in later after initial development
  - **Factor this into the budget**
  - Code that is designed to be testable is often better code



Create High Level Documentation

- Roadmap should include
  - System level diagram (hardware, physical connections, etc.)
  - High level software diagram (components and relationships between them)
- For classes create a VI tree which shows a simple example of how to exercise the API



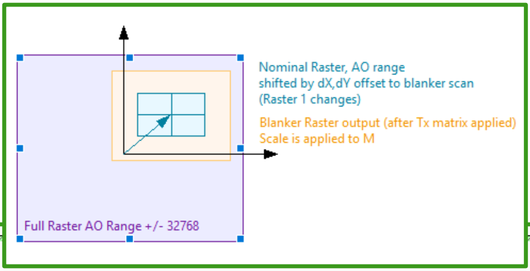


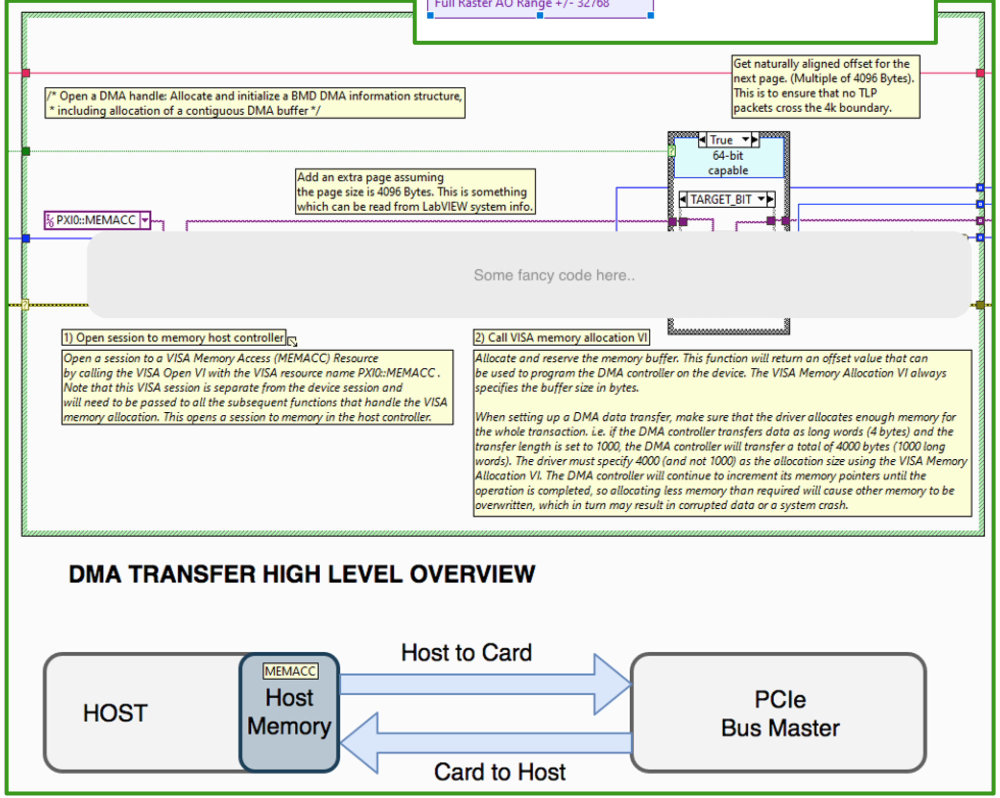
.. I have no memory of this place

Create Documentation in the Code

- Do not assume your code is so clear that it is self documenting, **it is not**.
- Write comments for another developer
  - Future you will also appreciate it!
- Document code where you will see it and keep it updated
  - **On the Block Diagram**

Put diagrams in your code if it is complex!


Nominal Raster, AO range
shifted by dX,dY offset to blanker scan
(Raster 1 changes)
Blanker Raster output (after Tx matrix applied)
Scale is applied to M
Full Raster AO Range +/- 32768



/* Open a DMA handle: Allocate and initialize a BMD DMA information structure,
* including allocation of a contiguous DMA buffer */

Get naturally aligned offset for the next page. (Multiple of 4096 Bytes). This is to ensure that no TLP packets cross the 4k boundary.

Add an extra page assuming the page size is 4096 Bytes. This is something which can be read from LabVIEW system info.

1) Open session to memory host controller

Open a session to a VISA Memory Access (MEMACC) Resource by calling the VISA Open VI with the VISA resource name PXI0::MEMACC . Note that this VISA session is separate from the device session and will need to be passed to all the subsequent functions that handle the VISA memory allocation. This opens a session to memory in the host controller.

2) Call VISA memory allocation VI

Allocate and reserve the memory buffer. This function will return an offset value that can be used to program the DMA controller on the device. The VISA Memory Allocation VI always specifies the buffer size in bytes.

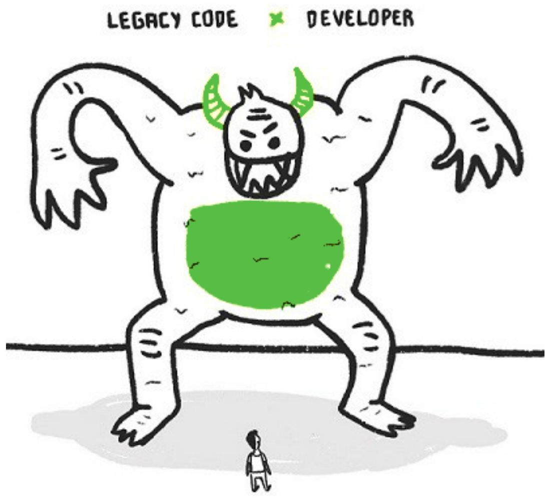
When setting up a DMA data transfer, make sure that the driver allocates enough memory for the whole transaction. i.e. if the DMA controller transfers data as long words (4 bytes) and the transfer length is set to 1000, the DMA controller will transfer a total of 4000 bytes (1000 long words). The driver must specify 4000 (and not 1000) as the allocation size using the VISA Memory Allocation VI. The DMA controller will continue to increment its memory pointers until the operation is completed, so allocating less memory than required will cause other memory to be overwritten, which in turn may result in corrupted data or a system crash.

DMA TRANSFER HIGH LEVEL OVERVIEW

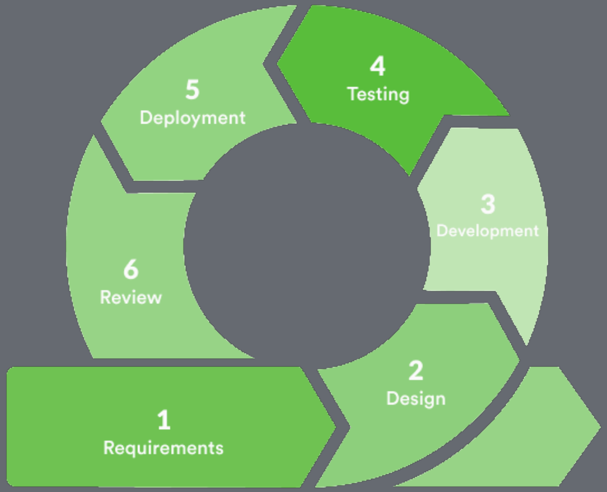
Summary Slide:

What we can do at the beginning of a project to avoid a bad legacy later on

- Keep the design simple
- Manage shared resources appropriately
- Plan on and add unit testing early
- Create documentation



II: Maintenance Phase



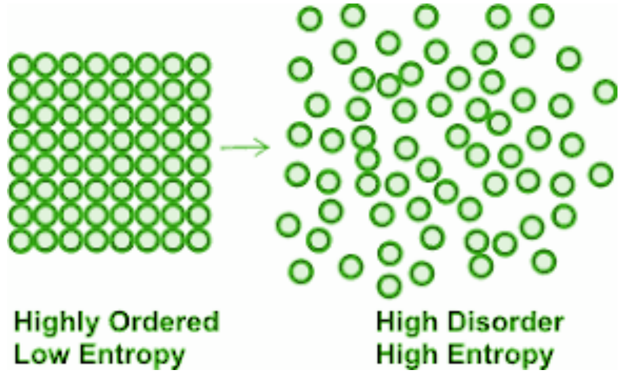
What causes software to get “worse” over the course of its life cycle and how can we prevent that from happening?

Code naturally becomes more complex as changes are made

Software Entropy

- Code complexity increases as changes are made and features are added unless it is actively managed.
- Increasing complexity increases development effort (and cost) to make further changes.

“Software becomes like biology over time, look at a sprawling forest or jungle and get that feeling looking at a codebase” - Michael Feathers



Refactoring can reduce software complexity

Boy Scout Rule: always leave the code behind in a better state than you found it.

- Also known as “opportunistic” refactoring. Living by this rule improves the quality of your code base over time.
  - Preparatory refactoring
  - Code improvements in the nearby vicinity

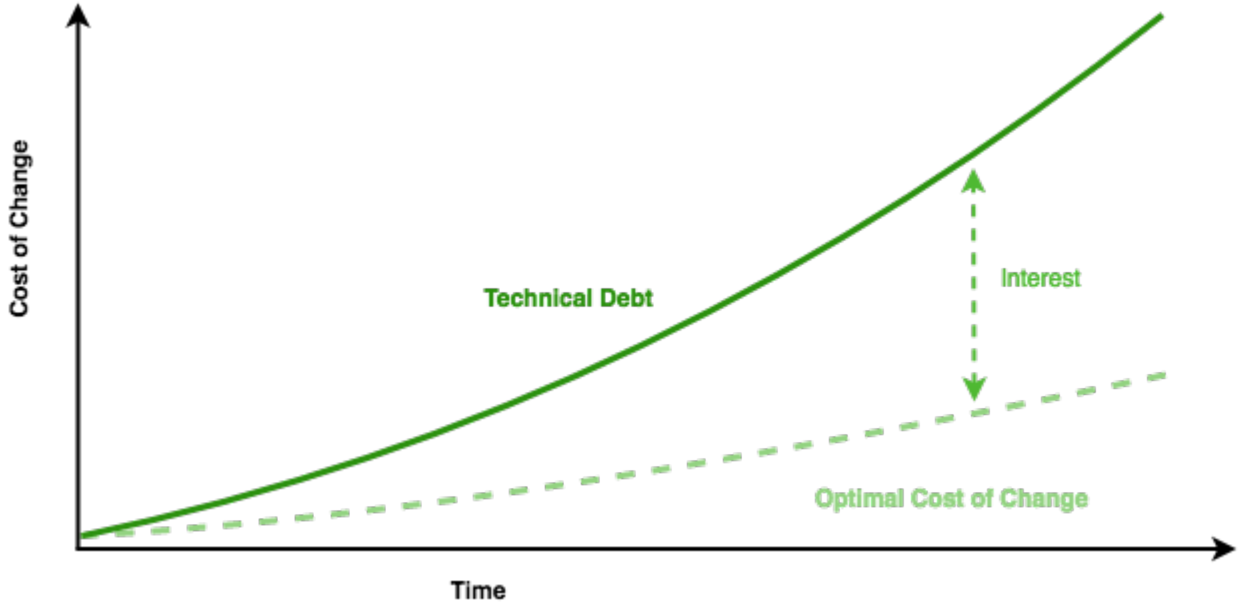
- Our code bases should get better over time instead of worse if we make refactoring part of our daily work!

“Always leave the code you’re editing a little better than you found it”
- *Robert C. Martin*

The cumulative effects of short term thinking

Technical Debt

- Metaphor introduced by Ward Cunningham back in the 1990s
- **Technical debt = future development time that is “borrowed” when shortcuts are taken to deliver quickly**



How can we manage technical debt?

- **Document technical debt**
  - When decisions are made to incur the debt, or when debt is discovered

- **Plan for and pay back technical debt**
  - Accrual is sometimes unavoidable (time to market pressures, unrealistic deadlines)
  - Allocate resources to cleanup and refactor code after a delivery

- **Track technical debt and communicate code readiness to stakeholders**
  - Consequences are often unclear to those responsible for the decision
  - **“Code readiness” should be a regular part of the conversation**

II: Maintenance Phase

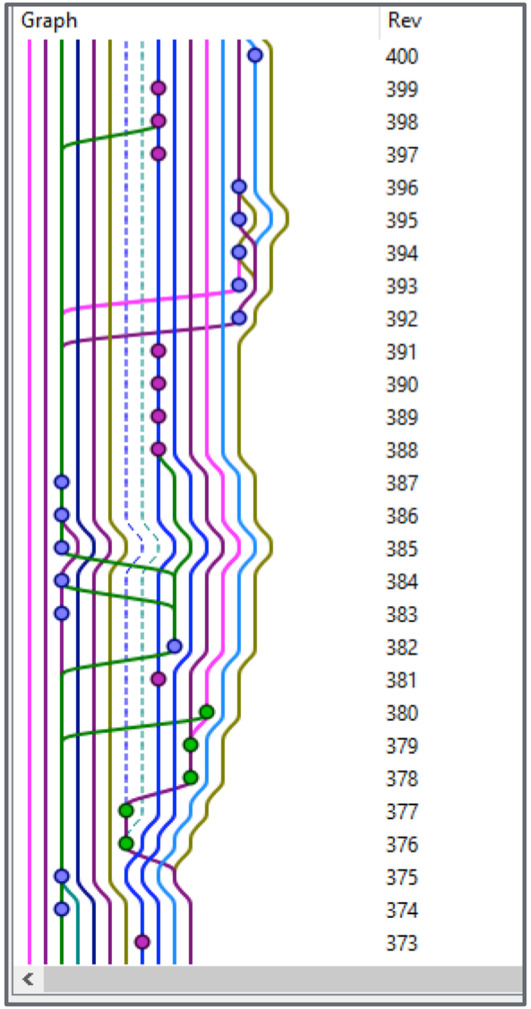
When GOOD code goes BAD

Specific examples of how code quality degrades and some prevention strategies.

When GOOD code goes BAD

Maintaining too many branches

- **Feature branches accrue technical debt**
  - Merges take time to resolve and can introduce bugs
  - More time open => more technical debt
  - Opportunistic refactoring is difficult with parallel development
- **Tips for managing parallel changes**
  - Split into smaller repositories (monolith -> microservices)
  - Avoid having feature branches open for too long
  - Document changes and allocate time for merges
    - Communicate if merge conflicts occur

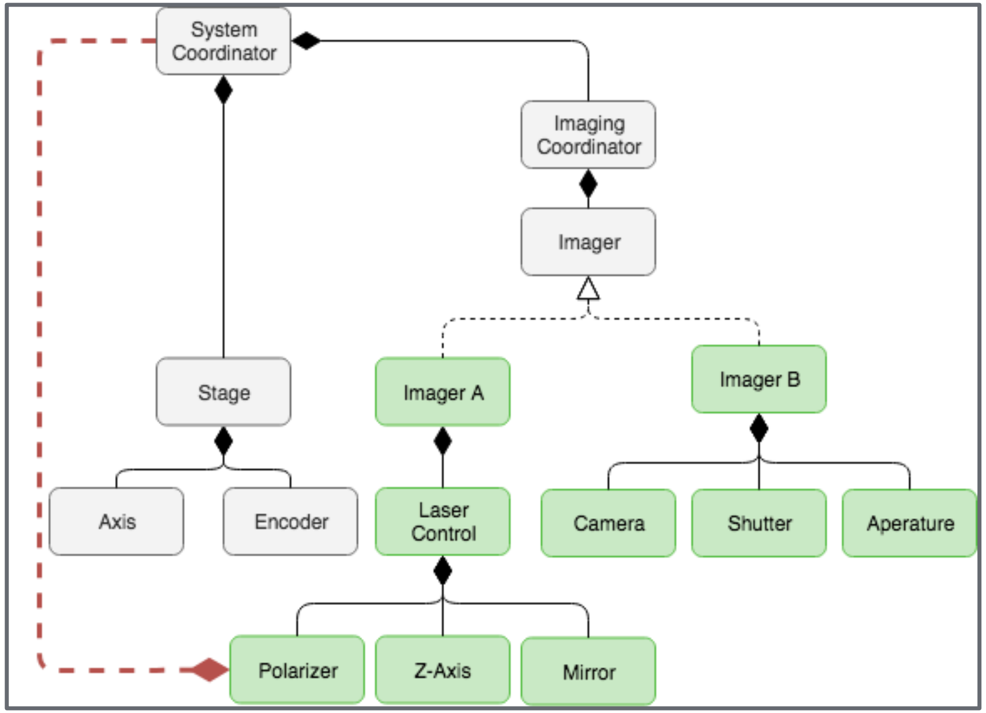


When GOOD code goes BAD

Unintended Coupling

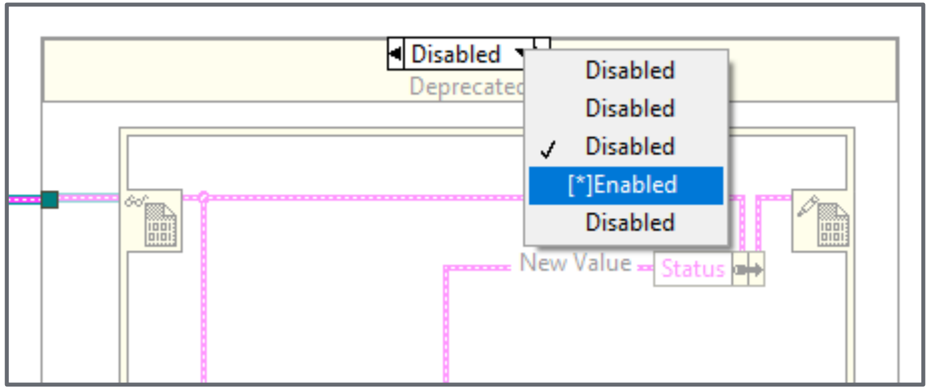
- **Developers working within a module may create bad dependencies without realizing it**

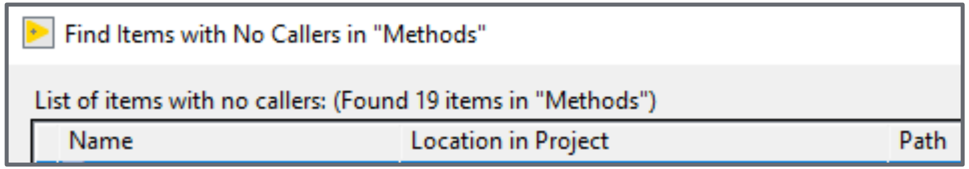
- **Your UML diagram is a Live Document! Revise and Review it often.**
  - The UML should reflect the relationships in the current software
  - Whose responsibility is it to update the UML?



Leaving Dead Code in your codebase

- **Dead code increases code complexity**
  - It takes more time to understand the code
- **Remove dead code!**
  - Code in your disabled structure
  - Unreachable Code
  - Unused Private Methods or Typedefs
  - Dead VIs on disk (not loaded in memory)
  - Unused protected or public methods & types




Find Items with No Callers in "Methods"

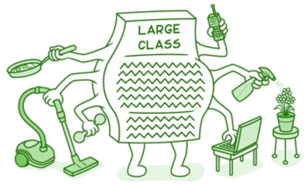
List of items with no callers: (Found 19 items in "Methods")

Name	Location in Project	Path

Warning! This VI is marked for deprecation

When GOOD code goes BAD

Evolution of the God Class



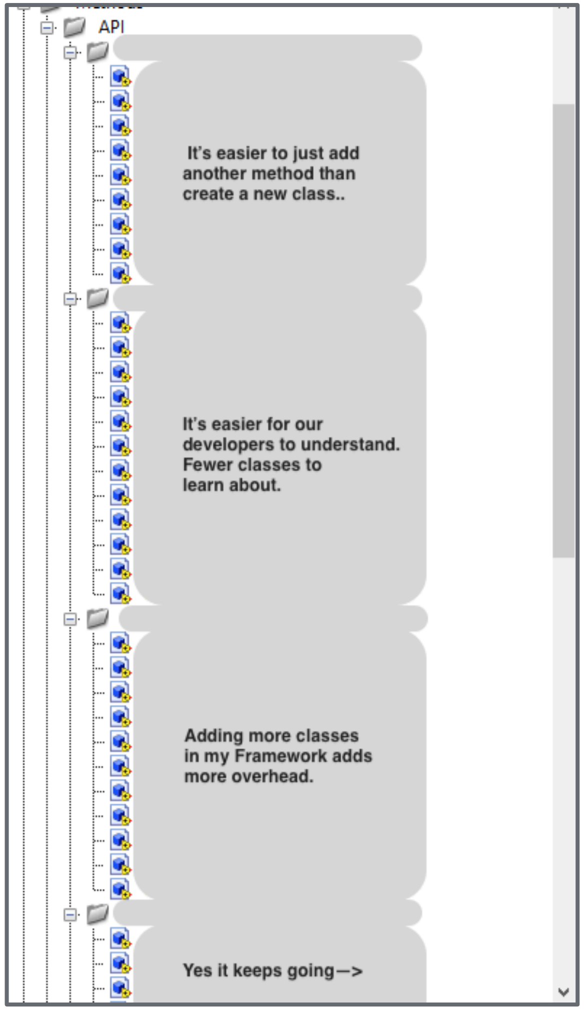
- Anti-pattern violates the **single responsibility principle**
  - .. Each module should be **responsible for a single functionality**.. all services should be narrowly aligned with that responsibility..
- Classes start small and grow over many feature requests

Proactive Solution:

- Features that do not align with class responsibility go in a new class.

Retroactive Solution:

- Large classes can be split into several smaller classes when bloating



It's easier to just add another method than create a new class..

It's easier for our developers to understand. Fewer classes to learn about.

Adding more classes in my Framework adds more overhead.

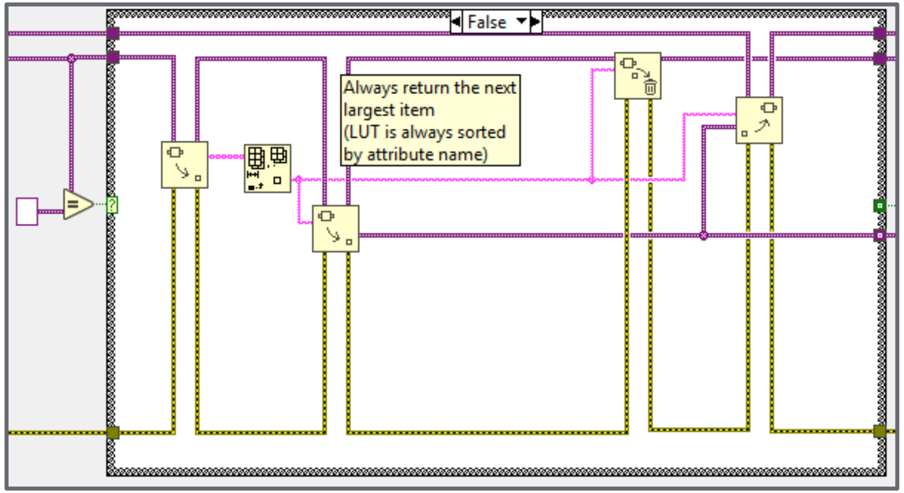
Yes it keeps going—>

When GOOD code goes BAD

Sacrificing maintainability for performance

- One of the reasons our code can get less readable and maintainable over time = performance optimization
  - At a minimum the block diagram may need more explanation
  - At the extreme, whole layers or dynamic linkage must be removed entirely

- **Add more detailed documentation after optimizing**

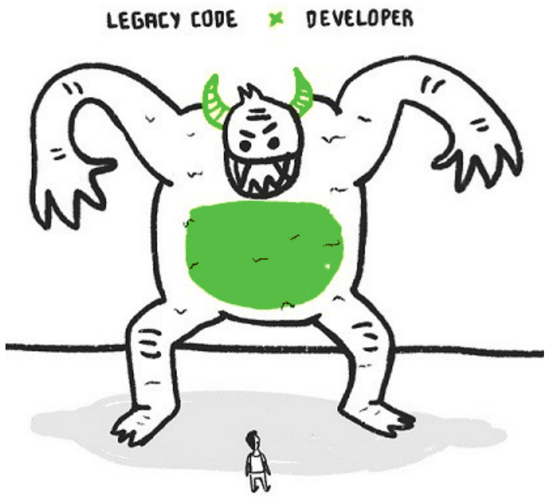

False
Always return the next largest item (LUT is always sorted by attribute name)

i.e. replacing a linear search with LUTs requires more context and documentation for variant attribute contents

Summary Slide

What we can do throughout the software lifecycle to avoid it becoming legacy

- Refactor as part of your ongoing and daily routine
- Document and plan for paying back technical debt
- Keep your UML up-to-date
- Remove dead code
- Document carefully when optimizing code



III: Successful Teams

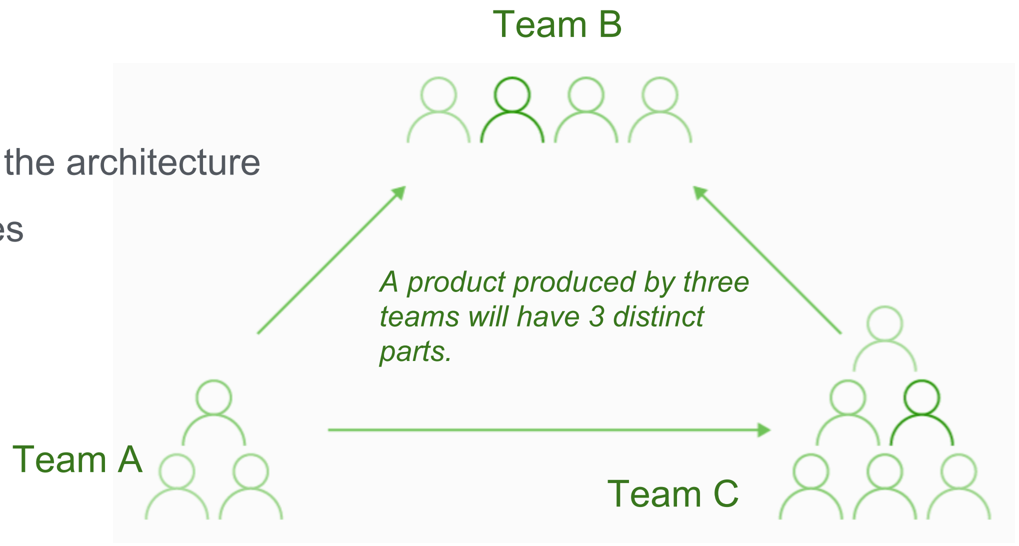
Non-technical related contributions to legacy software

Melvin Conway 1967

Conway's Law

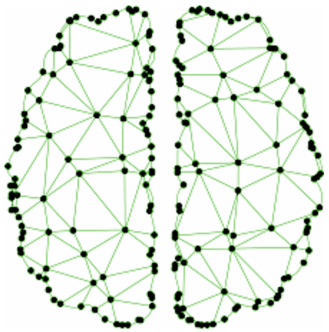
- *"Organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations"*
- In other words: A system architecture and the interfaces between components reflect communication and organization of the teams that developed the system.

- Improve communication between groups
- Organize development teams to complement the architecture
- Create smaller services/groups with strong ties


Team B
A product produced by three teams will have 3 distinct parts.
Team A
Team C

Losing Context

Institutional memory



- There is a reason **WHY** someone implemented the legacy code that way
  - Sometimes we refer to this as tribal knowledge of a code base
  - When people leave a company the context for design choices and history is lost

- What can we do to improve institutional memory?
  - Better documentation & creation of knowledge bases
    - Document high-level project details as well - what we did **and WHY**
    - Wikis, Creation of a short video library
  - Mentoring + Training
    - For continuity don't have one developer as the sole owner of all the design and details of one project

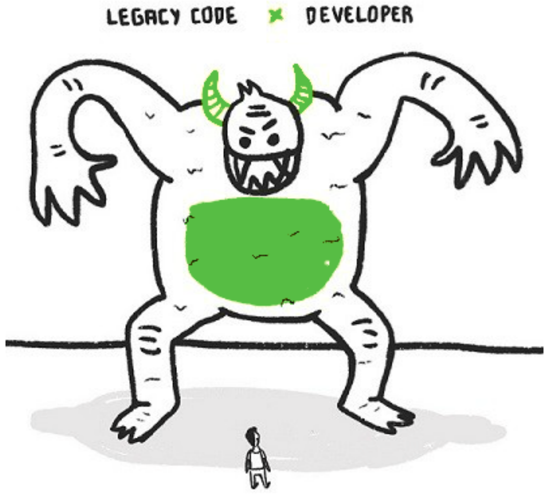
Successful Teams

- **Strive for continuous improvement**
  - Aim to incrementally make improvements to process and be open to trying new things and adjusting based on results
- **Create a culture of doing things “right”**
  - Encourage long-term thinking about what is best for the code base.
  - A culture that values quality will also encourage more pride in ownership over code-> leads to better maintenance.
  - Create clear code quality standards

Summary Slide

What can we do in our business and team culture to avoid a bad legacy

- Organize your team to compliment your architecture
- Improve institutional memory retention by documenting context
- Create a team culture of doing things properly



Key Takeaways

Let’s make a better legacy!!

Key Takeaways

- Active and continuous effort is required to improve our legacy, but it can be done!
  - **Prioritize maintainability in the initial design**
  - **Continuously work to maintain software throughout its lifecycle**
    - Refactor as part of the normal workflow
  - **Create a culture which values and prioritizes long term outcomes**
    - Communicate and manage technical debt
  - **Strive for incremental improvement**

Resources

Resources

- Tech Done Right Podcast: “Avoiding Legacy Code with Michael Feathers”
- Software Entropy Wikipedia on Software Entropy
- Refactoring large classes refactoring guru
- Legacy Monster Graphic Fernando Cejas: Technical Debt Guru Level Unlocked

Thanks!